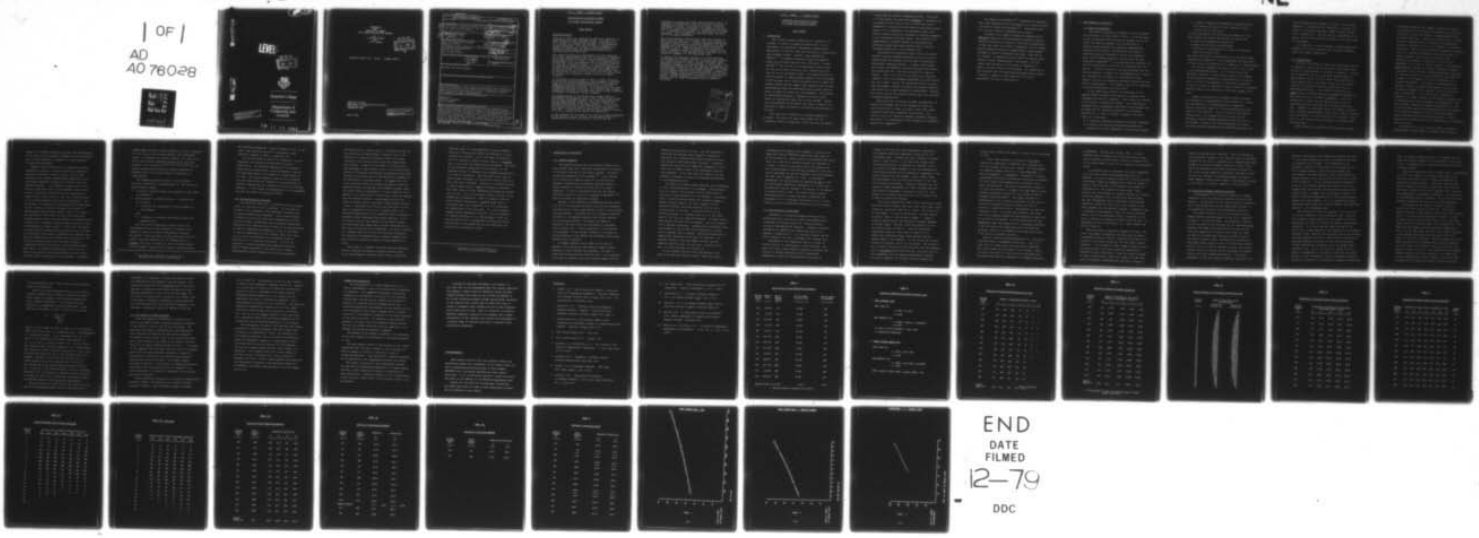
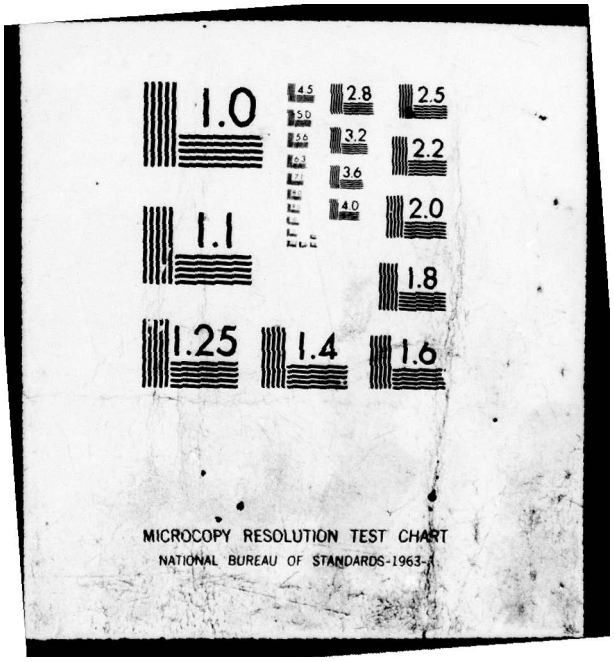
AD-A076 028 IMPERIAL COLL OF SCIENCE AND TECHNOLOGY LONDON (ENGLA--ETC F/G 9/2
COMPLEXITY AND COMPLEXITY CHANGE IN A LARGE APPLICATIONS PROGRA--ETC(
APR 79 G BENYON-TINKER , M M LEHMAN DA-ERO-78-G-017
UNCLASSIFIED R/D-2464 NL

| OF |
AD
A076028



END
DATE
FILMED
12-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

Imperial College

Department of Computing and Control

COMPLEXITY
AND
COMPLEXITY CHANGE
IN A LARGE APPLICATIONS PROGRAM

by
G. Benyon-Tinker
and
M.M. Lehman

Prepared under E.R.O. grant D-AERO-78-G017.

Imperial College,
Department of Computing and Control,
180 Queen's Gate,
LONDON SW7 2BZ.

April 1979.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Complexity and Complexity Change in a Large Applications Program

5. TYPE OF REPORT & PERIOD COVERED
Final Technical Report
29 Sep 77 — 28 Dec 78

6. PERFORMING ORG. REPORT NUMBER
R&D-2464

7. AUTHOR(s)
G. Benyon-Tinker &
M. M. Lehman

8. CONTRACT OR GRANT NUMBER(s)
DAERO-78-G-017

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Dept. of Computing & Control
Imperial College
180 Queens, Gate, London S.W. 7 2BZ

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
IT61102BH57-03

11. CONTROLLING OFFICE NAME AND ADDRESS
USA R&S GP (EUR)
Box 65 FPO NY 09510

12. REPORT DATE
APRIL 1979

13. NUMBER OF PAGES
46

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*
12 48

15. SECURITY CLASS. *(of this report)*
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

APPROVED FOR PUBLIC RELEASE

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES
The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Complexity
Software evolution
Large Programs

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
This report describes an investigation into the complexity of a single large banking application program, and into the changes in complexity which have occurred as result of the maintenance process The program, which is written in Algol, was developed to run as middleware on Burroughs computers. It has been in use for the last 5 years, during which time it has grown from a size of 36K source lines to 64K source lines by series of 13 releases, planned to occur at regular intervals.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 411449 UNCLASSIFIED

E.R.O. GRANT : D-AERO-78-GØ17

COMPLEXITY AND COMPLEXITY CHANGE OF A LARGE APPLICATIONS PROGRAM

FINAL REPORT

Extended Abstract

There are 2 objects in studying the nature of complexity of understanding in large software systems. The first is to gain insight into the underlying causes of complexity, so as to be able to minimise this ***ab initio*** by an appropriate design methodology. The second is define a measure or measures of complexity, by means of which degradation of system structure during the maintenance phase can be monitored and hence controlled.

This report describes an investigation into the complexity of a single large program, and into the changes in complexity which have occurred as a result of the maintenance process. The program, which is written in Algol, was developed by the Midland Bank to run as middleware on the Burroughs computers at their main computing centres. Its principal function is to schedule the nightly processing of their customer account records, although it has many other important capabilities. It has been in use for the last 5 years, during which time it has grown from a size of 36K source lines to 64K source lines by a series of 13 releases, planned to occur at regular intervals.

In conjunction with an attempt to give a careful discussion of the nature of understanding in large programs, a detailed study of the subject program suggested that one significant complexity factor was the breadth and depth of the fully decomposed functional structure. This concept does not appear to have been discussed previously in connection with program complexity, which has in the past been almost always considered in relation to relatively trivial programs.

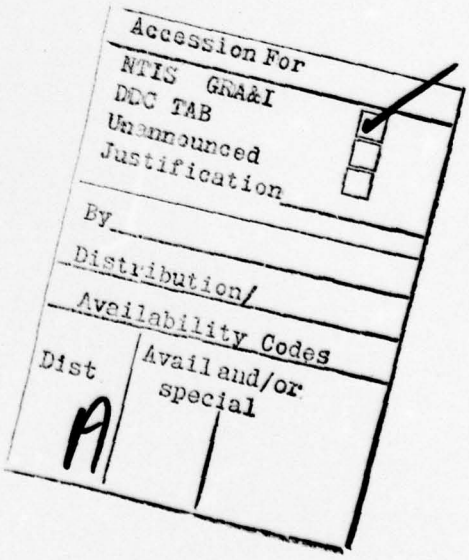
The program itself is constructed from a large number of procedures (650 growing to 1000), none of which is very large, so that the functional structure was closely approximated by the procedure calling hierarchy. This was extracted by a suitable modification to the Algol compiler. It was found that while the breadth of the calling tree at any level grew with program size, the shape of the tree was practically invariant. In particular, the maximum depth remained constant at 41 levels despite an 80% growth in program size.

It was observed that the size of the tree was largely determined by the multiple use of 'service' type procedures which in themselves had extensive sub-trees. Since it was not

reasonable to suppose that these sub-trees would need to be completely re-understood at every point where the parent procedure was called, the notion of 'distinct' calls was introduced. By this, a procedure only appears in the tree when it is called from a hitherto - unencountered procedure, ie. it represents a completely new and distinct use of the procedure. This suppresses all sub-trees after the first call of the parent procedure.

The tree of distinct calls did show some weak evolution in depth, from 16 levels to 18 levels, with a 54% increase in the total number of calls. A simple complexity measure was proposed in an attempt to capture the difficulty of understanding this tree, based on the qualitative discussion referred to above. After factoring out the evolution in size, a residual increase in complexity of between 15% and 30% was indicated. This appeared to be consistent with the subjective judgement of the programming team.

On applying the complexity measure to 2 of the principal sub-systems of the subject program, variations were observed on a release-by-release basis. In all cases where large variations occurred, it proved possible to relate these to significant changes which had been made to the program. This provides some confidence that the complexity measure defined is applicable to this type of program. However, it is clear that an equally important factor contributing to complexity in this program is the way in which procedures at all levels interact through shared global variables in a time-dependent manner. Further progress in this area will require the introduction of a multi-dimensional model of software structure.



E.R.O. GRANT : D-AERO-78-GØ17

COMPLEXITY AND COMPLEXITY CHANGE OF A LARGE APPLICATIONS PROGRAM

FINAL REPORT

1. Introduction

Even without a precise definition of 'complexity' in software, it seems to be generally agreed that it is an undesirable, but apparently inescapable, property which adds greatly to the cost, time and difficulty of contstructing large software systems. Further, large programs are observed [1] to undergo substantial change and expansion during the in-service phase of their life-cycle; a recent estimate [2] put the cost of the maintenance phase at more that 60% of the total life-cycle cost, compared with less than 40% for the development phase. The established fact of continual growth in large programs, together with a knowledge of the piecemeal way in which such growth tends to impact the original design concept, suggest that complexity may increase significantly during the maintenance process unless specific steps are taken to prevent this from happening. By corollary, an uncontrolled increase in complexity could be a major factor in limiting the useful life of such systems. (Useful life being defined as that period over which it remains cost-effective to adapt the system to the evolving requirements of the user.)

Thus there are 2 objectives in studying complexity in software. The first is to gain some insight into the fundamental causes of complexity, which could then be applied

to improving the original programming process. The second is to see whether any quantitative measures of complexity can be proposed, which could be used to monitor and control degradation of system structure during the maintenance phase. Implicit in this approach is the assumption that a cost-benefit analysis would show investment in reducing complexity to be a profitable life-cycle option. The financial (and organisational) interface between the development and maintenance phases may however prevent such an analysis from being carried out in an un-biassed way.

The project here reported on was of short (12 months) duration, and was intended principally as a pilot study to assess whether any progress could be made in developing a quantitative complexity measure. A single large program was studied, the EXECUTIVE scheduler developed and owned by the Midland Bank Ltd. This was selected for a number of reasons. First, it was felt to be sufficiently large and 'complex', and it had undergone significant evolution during the last 5 years. Second, the system was written in Algol, which encourages a regular structure, thus making complexity more likely to be simply definable and measureable. Thirdly, the source text for all released versions was still available in machine readable form.

There was little in the way of formal documentation to describe the program, and it was necessary to study the actual source text in order to acquire a good basic understanding of functional capability and structure. As a result of this, the complexity measures originally proposed were seen to be inappropriate to the program, and a different approach was suggested [3].

The compiler was modified [4] to generate the additional basic data required for the new measures, and a data reduction and analysis program was developed [5]. Using these tools the 13 extant version of EXECUTIVE were studied.

Although in the original proposal the emphasis was on measuring complexity, the investigators consider an equally important outcome of the project to have been the generation of some ideas about the nature of complexity, via the insight gained during the process of understanding a large program. Although it should be emphasised that these ideas are not yet complete or rigorous, they may serve as stepping stones towards a better understanding of the problem. For this reason, some discussion of the nature of complexity is given in the next section of this report. The reader who is less concerned with the underlying philosophy should proceed to Section 3 (page 13), which deals with the actual measurements and their interpretation.

2. Some Remarks on Complexity

2.1 Types of Complexity

The relative complexity of programs can be considered from basically 2 distinct points of view. On the one hand, we have the machine which executes the program, and on the other, the human who designs, uses and maintains it. Machine oriented measures of complexity (computational complexity) are typically concerned with the number of primitive operations required to execute the program on a given type of computing machine, and this area has been extensively researched [6]. Such measures are concerned only with the transformation of one set of numerical data into another. It does not matter to the machine whether the data have any meaning, or whether the transformations are 'correct'. All information in the machine is represented by pure numbers, which are in themselves completely abstract entities devoid of significance. (For example, there is no capability to associate dimensional units with data.)

To the human being, however, the usefulness of the program resides entirely in the ability to associate real phenomena with abstract variables: "Let U denote the velocity of the missile, in metres per second". Such specification statements are a fundamental constituent of the program, even although they are completely irrelevant to the machine execution of it. Thus, to design a program in the first place, it is necessary:

a) to represent observable phenomena by abstract variables;
b) to assert some functional relationship between the variables that holds independent of their actual values;

c) to define a sequence of mathematical operations which will produce the correct output response for each specific set of input data.

In order subsequently to understand the program that has been created, the reverse process must be carried out:

a) analyse the mathematical operations;

b) deduce the functional relationships;

c) interpret the relationships in terms of a conceptual model of the real world.

The total difficulty of constructing such a conceptual model represents the complexity of the program from the human point of view, and will be referred to as 'Complexity of Understanding'.

We see that complexity can arise at each of the 3 steps in the understanding process. At the first step we may encounter complexity in the detailed coding of the program, at the second step complexity of implementation of functions, and at the third step complexity of functional capability. It may not be easy to separate out these 3 effects in complexity measures based on the structure of the actual program source text.

Finally, we may need to distinguish carefully between the complexity in understanding a given program as it stands, and the complexity in understanding how to implement a desired conceptual change to it without affecting any other aspects of its behaviour. For as a result of the original design process, the implementation of a given function frequently contains embedded assumptions about the rest of the program. However, since these assumptions are fixed, once they have been determined the understanding process can

continue without further reference to them. In this sense, immediately a program has been understood, it becomes less complex. In making a subsequent change, however, we must seek to understand not only the obvious, local effect of the change, but also the consequential effect on the other parts of the program.

This requires an inverse approach to understanding the program, since we are concerned with arbitrary influences rather than logical dependencies.

2.2 Understanding

Before we can attempt to derive any quantitative measure of complexity, we must define as carefully as possible what we mean by 'understanding' in relation to software. A system can be understood (in the conventional English usage) at many different levels of detail, which range from having a rather general idea of the overall purpose of the system, to being able to explain the precise significance of each executable instruction. If we adopt the definition of a large system as one requiring many programmers organised on more than one level of technical management, it would be unrealistic to insist that one person should be able to understand the whole system at the individual instruction level. However, at some stage it must be possible to acquire such a detailed understanding, even if for a single individual this only relates to a restricted part of the system; otherwise it is not possible even in principle to determine what the system does.

Large systems are invariably constructed from a number

of program components (eg. segments, modules, procedures, sub-routines, etc.) which give the appearance of being self-contained, but which may interact through parameter lists and shared data areas. Usually, the system works by having a 'master' component which is entered at the beginning of execution, and which in turn calls a number of subordinate components. These in turn call further subordinate components, and so on, creating a hierarchical calling structure. In complex programs, this structure can be very deep; in EXECUTIVE, for example, it extends to over 40 levels. Regardless of the level at which it appears in the hierarchy, each component consists of 'visible' executable statements interspersed with statements invoking subordinate components. In order and content, the visible statements constitute the framework of the component, and must be understood as a co-operating sequence. The process of understanding the sequence involves a statement-by-statement analysis, with the gradual emergence of an abstract functional picture of what is being done. At any point where a subordinate component is invoked, analysis of the original component must be suspended while the new component is investigated. When this has been completed, an abstract description will exist in the mind which must then be assimilated into the analysis of the original component, where it will either expand or modify the partial picture previously generated. Thus to understand any component entails the integration of 2 different levels of detail.

Consequently, it is not strictly correct to speak of understanding a program down to a certain level of detail,

because it is always necessary to achieve some understanding at the level of the individual statement, and the functional power of that statement is limited to a simple operation on primitive data objects.

However, once the functions of a component have been understood in detail, it should be possible to generate a conceptual description of what it does. Part of that description will embody corresponding descriptions of any subordinate components invoked by it, and so on down the calling hierarchy. It is tempting to suppose that if we stop at a particular level in the hierarchy, then we shall be able to generate a description of the system that is complete down to that level, ie. that we can equate levels in the calling hierarchy with levels of description. The correctness, or otherwise, of this supposition is absolutely crucial to the way in which we approach the problem of understanding large software systems. When it is correct, the implication is that there is very little functional coupling between levels, and that details in the descriptions of lower level components rapidly become irrelevant as they are assimilated at higher levels. In this case, the amount of mental effort required of any one person at any one time is restricted to that needed to understand one component only. When the supposition is not correct, the implication is that the mental grasp required of an individual extends to integrating detailed descriptions of sub-components across many levels. We believe that the latter case is the one which usually applies with large systems, and that this is the basic cause of complexity in such systems. (A specific

example might be where a control variable is set by a component at a low level in the calling hierarchy, and subsequently tested - to determine the course of action - at a high level.) We note that several complexity measures which have been proposed in the literature [7],[8],[9],[10],[11],[12] do not address systems where interaction between components at different levels is the major source of difficulty of understanding.

The foregoing discussion leads to the following, tentative, definition of 'understanding' in the context of large software systems:

I. A system is understood when each component of the system is understood;

II. Each component is understood when it is possible for an individual

a) to explain the significance of each statement in the component,

and,

b) to generate a conceptual description of what the component does.

Note that II(b) automatically implies that we are considering the role of the component in relation to the system as a whole, by virtue of requiring a conceptual description. Note also that since this definition does not specify that each component must be ‡ Understood by a separate individual, it applies to both the weakly-coupled and the strongly-coupled types of system described above.

---

‡ We shall use a capital U to denote use in the sense which has just been defined.

The difference between the 2 types of system is that in the latter one individual may have to Understand several components together in order to Understand one.

We can now proceed to define the 'complexity' of each component as the amount of mental effort required to Understand it. By assessing complexity at the component, rather than the system level, we are attempting to capture the irreducible mental load imposed on a single person at one time. The system complexity needs now to be assessed in a more qualitative way, possibly by the distribution of component complexities as a function of level in the calling hierarchy; at this stage, the organisation of the programming team may also enter as an important factor.

2.3 The Sub-Division of Systems

The system as a whole has 2 interfaces with the external world, an input and an output interface. If the system is divided up into components, then additional interfaces will be created between these components. Across these internal interfaces will flow data which will predominantly not have a direct external significance, but an internal significance depending on the functional path by which it has been generated. In order to Understand any component, it will be necessary to know the significance of each variable used, since otherwise the functions performed by the component cannot be abstracted into a conceptual description. (This is not applicable to 'pure' procedure-type components written in terms of formal parameters, since these are, by design, Understandable without reference to the actual parameters:

we treat these as a special case.) It follows that part of the difficulty of Understanding a component lies in the difficulty of attaching a meaning to each of the input variables in the context within which the component has been entered. It may not be possible to do this without some knowledge of that context, and it is this factor which prevents the classical hierarchical decomposition from being carried to the absurd limit of allocating each statement to be Understood by a separate person. We suggest that this is one important reason why software systems are orders of magnitude harder to understand than hardware systems, which can almost always be decomposed to a fairly primitive level. However, it is instructive to reflect that limits are reached, even for hardware systems. A good example would be the humble bi-stable circuit (flip-flop), whose overall function is easy to understand as an entity, but which would be very hard to understand if split into 2 separate parts; the functional significance of the interface cannot be precisely described without considering both parts together. It is of interest that the vast majority of electronic systems are constructed from simple basic circuits consisting of no more than half-a-dozen active components. The basic 'circuits' of a software component are not the individual statements, but sequences of statements which co-operate together, and these can be as long and as varied as human ingenuity can devise.

Input data to a component can arise from many different sources, spanning the hierarchy between variables global to the whole program (including on-and off-line files) and

variables local ‡ to the immediately invoking component. There are 2 factors which affect the difficulty of understanding each variable. One is the number of places at which the variable is given a value, since it is possible that at each place the significance is different. The other is the functional complexity of the variable, ie. the dependency chains of statements and variables through which its value was determined. The first factor will tend to affect global variables, the second will tend to affect the most local input variables. Further, if a variable is given a value at many places, it may be necessary to determine the actual place or places immediately preceding the current point of use. That is, the global control structure of the program may be significant. What this means is that the person trying to Understand a component might have to investigate many other components in order to appreciate the significance of the input data. In a well- designed program, the data-dependency structure would correspond to the calling hierarchy, but in real programs it will not, and this gives rise to incompatibility between the formal sub-division of the program and the true functional sub-division of the problem it solves.

---

‡ Obviously, one is thinking here in terms of block-structured languages.

3. Measurements on EXECUTIVE

3.1 General Approach

The original proposal for this research project was to test whether there was a detectable upward drift of variables in the block structure of the subject program. Following a detailed examination of the program [3], it was decided not to pursue this approach since all the functionally important variables were necessarily declared either as global variables or within the outer block of a major component: it was not anticipated that the null result previously found for the first 4 releases of EXECUTIVE [13] would be reversed in subsequent releases.

However, it was observed that EXECUTIVE had been written as a set of inter-acting procedures, and that the functional structure was, to a first approximation, based on the procedure calling hierarchy. Following the line of argument developed in Section 2, it appeared plausible that several features of this calling structure - the depth, the degree of branching, the total size, the distribution of procedure calls by level, etc. - could be relevant to the complexity of the program. It was decided, therefore, to analyse the calling structure to see whether any detectable evolutionary pattern could be observed.

A standard compiler option (XREF) can be used to generate a file giving the card sequence number of every reference to every variable and procedure. In order to obtain the calling structure from this file, it is necessary to be able to associate every card sequence number with the

procedure within which it occurs. This was achieved by modifying the Burroughs Algol compiler to produce an additional file containing the sequence numbers at which each procedure declaration began and ended. A reduction program was then written to merge the information in the 2 files and to create a table of the procedures called by each procedure. At the same time, the compiler was also modified to produce an analysis of procedure declarations by lexicographical level.

The table of procedure calls represents the hierarchical, or tree, calling structure of the program. Each procedure call is a node in the structure, and the nodes which it calls are in turn connected to it by branches. A procedure which is called from many places will therefore appear at several distinct nodes in the tree, together with its dependent branches, and these distinct nodes can be at varying levels in the tree. A procedure which calls no further procedures is a terminal node (i.e. at the lowest level of the tree, all nodes are terminal). At other levels, some fraction of nodes will be terminal. The tree structure can be analysed recursively: starting from an arbitrary node, one branches in turn to each of the nodes which it calls, until a terminal node is reached. (Recursive chains must be detected and terminated at the point of recursion.) By keeping a record of the total number of nodes, and the number of terminal nodes, at each level, a profile of the tree, or of any sub-tree, can be built up.

In the original version of the tree analyser, each node in the tree was actually visited. While this worked

satisfactorily for smaller test programs, it proved to be impossibly slow for EXECUTIVE, where there were found to be of the order of 1 million nodes at the maximum width level. (The C.P.U. time for a complete analysis was estimated at 13 hours.) Since this was mainly due to the repeated use of a limited number of service-type procedures, it was overcome by recording the sub-tree dependent from each procedure the first time it appeared at a node. At subsequent appearances, the sub-structure was appended in toto to that of the procedure immediately antecedent to it in the tree. The execution time was reduced to 30 seconds, at the expense of taking up 300K of core. The actual results obtained from analysing the 13 extant versions of EXECUTIVE are discussed in § 3.3, following a discussion in § 3.2 of the overall evolution of EXECUTIVE between 1973 and the present.

3.2 The Evolution of EXECUTIVE

EXECUTIVE is a scheduling program developed by the Midland Bank Ltd., and used to control the nightly batch-processing of customer accounts at 2 computer centres, running on Burroughs B7700 computers. A general functional description of EXECUTIVE has already been given in an Appendix to reference (3). It is written in Algol, and was first released to the user in November 1973 (release 216).

Enhancement of the program has been by a series of planned releases at approximately 5 month intervals. The work to be done in each release is planned in advance, and consists of changes negotiated with the operational departments to meet the evolving needs of the Bank or

changes in the law, plus other improvements requested by the computer operations department or generated by the programming team. Although the latter category of enhancements is under control, in the sense that changes which are obviously going to be difficult to implement tend to be avoided, the former category is not, and represents an evolutionary pressure that is potentially independent of the current structure of the program. Often, indeed, these new requirements could not have been foreseen at the time of the original development. Generally speaking, only one release is under development at a time, although there was some parallel development of releases 221 and 222; also release 228 contained items left out of 227 because it had to meet an operational deadline.

The initial release (216) had a size of 36K Source lines, while the current release (228) has a size of 64K lines, which represents a growth of some 78% in 5 years. The actual size of the program - in lines of source text - at each release is given in Table I, and plotted in Figure 1. Figure 2 shows the growth against the alternative (pseudo) timebase of release number. The best linear and quadratic fits to the data are given in Table II. From the correlation coefficients obtained, there does not appear to be a significant difference between using calendar time and release number as the timebase. In view of the way in which releases are planned, this is not unexpected. The quadratic fit is slightly better than the linear fit, and both timebases show an <u>increasing</u> growth rate with age. Since the size of the programming team has remained approximately constant, this result suggests (if anything) that the program has

become easier, rather than harder, to work on with the passage of time.

A very rough estimate of the level of programmer productivity applicable to EXECUTIVE can be derived from the size increase over the last 5 years. The team strength has remained roughly constant at 1 Chief Programmer + 2 Senior programmers + 2 programmers. Allowing for training requirements and other non-productive activities, we estimate the total effort from November 1973 to November 1978 at about 200 man-months. This gives a productivity figure of 140 lines per man-month, a figure in the same region as that obtained by Walston and Felix [14] in a study of some 60 programs during development. The figure obtained is an underestimate, since it makes no allowance for the lines of source text replaced during the process, and this is certainly a significant factor. Unfortunately, no data are available on this. If we make the assumption that every line of the original program has been replaced at least once in the life of the program - and from discussion with the programming team, this appears to be a reasonable assumption - then the productivity figure goes up to 320 lines per man-month, which is, if anything, on the high side relative to the results of Walston and Felix.

However, lines-of-source-per-man-month is a notoriously unreliable index of programmer productivity. Also, it is known that the changes which have been made to EXECUTIVE have varied widely in their impact on the program. Hence, it would be most unwise to draw any firm conclusions from the growth data about the complexity, or changing complexity

of EXECUTIVE. The best that can be said is that EXECUTIVE does not appear to be either very easy or very hard to enhance, and that there is really no evidence that it has become harder to work on as a result of its substantial growth.

An alternative way of measuring the size of EXECUTIVE is by the total number of procedures from which it is composed. This data is given in Table III, and the relationship between number and program size is shown in Figure 3. Overall, there has been a 47% increase in the number of procedures for a 78% increase in source lines. Note, however, that the increase in procedure numbers is a function of the lexicographic level at which they were declared, and that lexicographic level 2 procedures (those declared immediately within the outer block, and so global to the whole program) show the smallest increase. Remembering that a procedure at lexicographic level (n+1) must be declared within, and is only accessible within, a procedure at lex. level n, one way of looking at the data is by the ratio of procedures at (n+1) to procedures at n. These figures are given in Table IV.

These data may be interpreted roughly as follows. The level 2 procedures correspond to the principal sub-system of EXECUTIVE, plus any 'service' functions which need to be available to the whole system (eg. the File Access Methods component). The 25% growth in these suggests there has been an overall growth in the basic functional capability of EXECUTIVE of about this amount. Since the overall increase in source lines is over 3 times this figure, we infer that much of the growth of EXECUTIVE has been due to changes in

detail to existing (sub-) functions. This is reflected by the substantial increase in the relative number of procedures declared at lex. level 5, together with a comparatively small change in the relative numbers declared at levels 3 and 4. The final inference is that there has been an increase of 65% in the average length of a procedure, although we cannot say how this increase is distributed amongst the levels. Taken together, these results suggest a considerable increase in the local, or internal, complexity of each procedure.

3.3 Functional Program Complexity Analysis

As discussed in §3.1, our current approach to program complexity is via the functional hierarchy of the calling sequence. The objective of the actual analysis of EXECUTIVE was to see whether there was any significant evolution in any parameters relating to that hierarchical structure.

The overall evolution of the tree calling structure between release sequence numbers 216 and 228 is summarised in Table V. The most striking - and unexpected - feature is that the depth of the structure has not increased at all. Secondly, the shape of the distribution of nodes versus level has changed only in detail, even though the number of nodes at any given level has gone up twice as fast as the number of source lines: the mode of the distribution oscillates between levels 19 and 20, and the median between levels 18 and 20, when all releases are taken into account. Thirdly (Table VI), neither has the percentage of terminal nodes at any level changed significantly. The initial assessment is that this implies tremendous stability in the

functional structure of the program, and supports the view [4] that large programs have a kind of infrastructure which is extremely hard to change. This infrastructure is certainly a function of the nature of the application and of the design approach, rather than just program size. (For example, the Algol compiler, which was subjected to a similar analysis, had a size of some 35K lines with a total of 693 procedures. Yet the calling hierarchy descended to level 53, with mode and median both located at level 30, somewhat more than half-way down the tree.) We produce some evidence later to suggest that this apparent stability is partly a statistical phenomenon due to the smoothing effect exerted by a large program. But it is nevertheless highly significant when viewed over so many releases, and occasioned surprise in the programming team.

The rationale for the above approach is that each node in the tree represents a new intellectual task. Since the total number of nodes exceeds the total number of distinct procedures by a factor of 40,000, this seems a questionable assumption. It is clear that the depth and span of the tree is due in great measure to the repeated use of procedures, especially those 'service' type procedures which have extensive sub-trees dependent from them. Unfortunately, these service procedures generally interact with the calling procedure through shared global variables as well as through formal parameter lists. Thus it is not safe to assume that their effect is independent of the point at which they are called. However, it may be possible to make a compromise assumption by which the sub-tree of a procedure is regarded as Understood

after it has been analysed just once: it remains only to check the visible statements of the procedure at each point of call to determine how it interacts functionally with the calling procedure.

Under this assumption, which is almost the most simplifying one which can be made, we can ignore those nodes in the tree which are there solely by virtue of being part of a previously analysed sub-tree. (Obviously, each sub-tree must be completely analysed at least once; when this has been done, a marker can be set to indicate that it need not be re-analysed at a subsequent occurrence.) We shall refer to the remaining nodes as 'distinct', in the sense that they represent completely distinct uses of a given procedure.

The analysis of the number of distinct nodes as a function of level is given in Table VII. Apart from a drastic reduction in the number of nodes at any level (except level 1), the most obvious difference to Table V is that the depth of the tree is reduced from 41 to 16, with some slight evolution to greater depth (18) by release 228. This means that the longest distinct sub-tree in EXECUTIVE is only 16 to 18 levels deep, and that all the nodes occurring in the complete tree below there are due to the use of chains of 'service' procedures. The total number of distinct nodes has grown by 54%, slightly faster than the growth in the total number of procedures (47%), and slower than the growth in source size (78%). This implies that the average number of distinct calls per procedure - the number that would be deduced from a static analysis of the source text - has gone up by 5%, while the density of calls per source

line has declined by 14%. Thus neither of these measures shows any strong trend.

Based on the ideas presented in §2.2, we can define a complexity measure which attempts to quantify the mental effort needed to integrate the functional decomposition into a complete description. If we suppose that the difficulty of integrating a low-level procedure is some function of the depth at which it occurs, we can define

$$C_r \equiv \frac{\sum_{i=1}^{m} n_i (i)^r}{\sum_{i=1}^{m} ni}$$

where n_i is the number of nodes at level i, m is the maximum depth of the tree, and r is a power-law index. The denominator is a normalising factor: C_r is essentially just the r'th moment of the distribution of levels. The behaviour of C_1, C_2, C_3, and C_4 is given in Table VIII. We note, first, that there has been a modest increase in all 4 coefficients. The increase is itself a montonic function of r, and is over and above any effect due just to the increased size of the program. Secondly, we note that the evolution in C_r is almost exactly zero up until release 223; for all C_r, the changes take place entirely between releases 223 and 228. It has not so far proved possible to find a convincing explanation for this behaviour, although the sharp increases at release 225 and 227 do coincide with substantial changes in EXECUTIVE imposed by new operational requirements.

Since the increase in sheer size of the source text has been the outstandingly dominant factor in the evolution of

EXECUTIVE, it is difficult to assess the extent to which the complexity of the program as such has changed. The impression gained from the programming team is that there has been an increase in the overall difficulty of understanding EXECUTIVE, but that it has not been large. On this basis, we believe that a value for r of between 2 and 3 would give a complexity coefficient C_r whose behaviour was consistent with this very qualitative judgement. This indicates a complexity increase of between 15% and 30%.

3.4 The Analysis of Some Sub-Trees

Since the analysis of EXECUTIVE as a whole had not shown up any marked trends in possible complexity measures, it was decided to look at some of the sub-systems. The principle sub-system is associated with the WS procedure, which schedules and initiates the applications programs. In addition to the overall trend, it was noticed that there was a sharp increase in the depth of the complete sub-tree, from 28 to 35 levels, between releases 26 and 27. On enquiry, it was suggested that this might have been due to the changes made to the File Access Methods component at release 227. The analysis of FAM and its sub-tree provided conclusive evidence that this was the case: the relevant figures are given in Table IX. (This result is slightly meretricious since FAM is not used directly by WS, but appears in its sub-tree because it is passed as a parameter to the external applications program.)

Another important sub-system is the operator communications component, OPSCOM. The behaviour of the OPSCOM sub-tree is given in Table X. Note the marked changes at releases

218, 220 and 225. Unfortunately, it has not been possible to find out what happened at releases 218 and 220. However, it is known that at release 225, OPSCOM was substantially altered in order to accommodate a new table structure imposed by another change, and it is claimed that the opportunity was taken to simplify the component. This claim appears to be borne out by our analysis. The functional capability of OPSCOM was increased, yet the total number of distinct nodes was more than halved and C_2 reduced by 25%. This is the only direct evidence we have that there is a correlation between the structural parameters of the calling tree and the perceived complexity of the program.

The investigation of these 2 sub-trees has shown larger variations in the tree parameters than were evident from the analysis of EXECUTIVE as a whole. We interpret this as stochastic smoothing, due to the effect of combining a number of independent changes in a large program volume. It follows that any obvious variation in the parameters for the whole program should reflect a major change to its complexity. If we examine Table VIII in more detail, we see that 'obvious' variations in C_2 and C_3 occurred at releases 225 and 227. Both of these releases were associated with significant extensions to the functional capability of EXECUTIVE, the former impacting about 75% of the program source text.

4. Summary and Conclusions

The investigation of a large program has led to a better understanding of some of the factors which contribute to complexity. This suggested an approach to measuring complexity by considering the structure of the procedure calling tree, and some special software was developed to extract this structure from the program source text. A complexity measure has been defined, and the analysis of the last 13 releases of the subject program reveals a modest evolution in this measure in addition to the overall increase in the size of the tree. The analysis of some sub-trees supports a conclusion that the complexity measure and the tree size are indeed correlated with the complexity of the program, as assessed by subjective judgement.

The absence of any dramatic increases in the complexity of the whole program is attributable to the smoothing effect of size.

The most dominant factor in the evolution of the subject program has been the sheer increase in size, although this does not appear, in itself, to have been a cause of increasing complexity. This conclusion is possibly moderated by the experience of the programming team with the subject.

The overall conclusion is that considerable progress has been made in understanding complexity in a large program, and that a useful quantitative measure of complexity has been demonstrated. Complexity however, is not a single simple concept and many further facets of this property remain to be investigated.

Although not mentioned elsewhere in the report, it was clear that to the programming team, who already Understood the program, a major source of difficulty during the maintenance process arose from the way in which procedures at all levels interacted directly through shared global variables. It was observed that a great deal of effort was spent in trying to determine which variable was set by what procedure and at what point in time. This is a distinct, but equally important complexity property of the program, and a proper continuation of this research will require the development of a suitable model of software structure to represent these interactions adequately.

Acknowledgement

This project would not have been possible without the substantial support and co-operation of the Midland Bank Ltd., both in granting unrestricted access to their program EXECUTIVE and in providing computing and other support facilities. It is a particular pleasure to record the active help given by the members of the EXECUTIVE programming team.

Support was also given by the Burroughs Corporation, who kindly gave access to some of their proprietary software for the purposes of this project.

References

1. Lehman, M.M: "Laws of Evolution Dynamics - Rules and Tools for Programming Management." Infotech Conference "Why Software Projects Fail", London, April 1978. (To be published in Conference Proc.)

2. Wolverton, R.W.: "Software Life Cycle Management - Dynamics Practice: Summary". Second Life Cycle Management Workshop, Atlanta, August 1978, page 27.

3. Interim Report No.1, ERO Contract DAERO-78-G017. "Complexity and Complexity Change of a Large Applications Program". Imperial College, April 1978.

4. Ibid, Interim Report No.2. July 1978.

5. Ibid, Interim Report No.3. October 1978

6. Hartmanis, J. and Hopcroft, J.E.: "An overview of the theory of Computational Complexity". J. Ass. Comp. Mach., 18, 444.(1971).

7. Halstead, M.H.: "Elements of Software Science." Elsevier North-Holland, New York, 1977.

8. McCabe, T.J. "A Complexity Measure". IEEE Trans. Soft. Eng., SE-2(4), 308, (1976).

9. Sullivan, J.E.: "Measuring the Complexity of Computer Software". Mitre Corp. report MTR-2648, Vol. V, June 1973.

10. Van Emden, M.H.: "The Hierarchical Decomposition of Complexity". Machine Intelligence, 5, 361. (1970)

11. Ferdinand, A.E.: "A Theory of System Complexity". Int. J. of General Systems, 1(1), 19-33. (1974)

12. See also: Int. J. of General Systems, 3(4), (1977), which was devoted to papers on System Complexity.

13. Hooton, D.H.: "A Case Study in Evolution Dynamics". M.Sc. thesis, Department of Computing and Control, Imperial College. September 1975.

14. Walston, C.E. and Felix, C.P.: "A Method of Programming Measurement and Estimation". I.B.M. Sys. J., 16(1), 54-73. (1977).

TABLE I

The Evolution of Some EXECUTIVE Size Measures

Release Version No.	Release Date	Age at Release (days)*	Size by number of source lines	Size by number of Procedures
216	6/11/73	141	35 845	657
217	5/ 2/74	232	37 362	684
218	27/ 8/74	435	39 867	718
219	8/ 4/75	659	42 946	749
220	6/11/75	870	44 345	752
221	28/ 4/76	1045	45 322	772
222	20/ 5/76	1067	47 560	779
223	10/ 9/76	1180	47 645	783
224	8/ 8/77	1513	51 758	811
225	24/10/77	1597	57 662	884
226	10/ 7/78	1856	60 404	894
227	2/10/78	1940	62 838	958
228	20/11/78	1989	63 843	967
Overall growth, 216 to 228 :			78.1%	47.1%

* From the release of version 214 on 18/6/73

TABLE II

Analysis of EXECUTIVE Size Growth by Source Lines

1. Real (Calendar) time

Best linear fit:

$$s = 32563 + 14.379\ A$$

$$r = 0.982$$

Best quadratic fit:

$$s = 35810 + 6.0539\ A + 0.0039818\ A^2$$

$$r = 0.992$$

(s = size in lines of program, A = age in days

r = correlation coefficient)

2. Pseudo (release number) time

Best linear fit:

$$s = 32153 + 2411.1\ RRN$$

$$r = 0.984$$

Best quadratic fit:

$$s = 34905 + 1310.4\ RRN + 78.62\ RRN^2$$

$$r = 0.990$$

(RRN = relative release number = release number - 215)

TABLE III

Evolution of Procedure Declarations by Lex. Level

Release Version No.	Number of Procedures Declared at Level						
	2	3	4	5	6	7	8
216	141	179	238	61	32	6	
217	143	184	246	68	35	7	1
218	145	191	271	83	21	6	1
219	155	199	275	91	22	6	1
220	155	203	274	91	22	6	1
221	157	213	269	90	36	6	1
222	167	213	266	89	37	6	1
223	165	207	276	91	36	7	1
224	164	211	280	125	29	2	
225	174	256	284	141	35		
226	171	242	291	151	39		
227	175	258	328	154	43		
228	176	260	333	156	42		
Overall Growth (%):	24.8	45.2	39.9	155.7	(Level 6 and below) 25.0		

TABLE IV

Evolution of Relative Procedure Frequencies

Release Version No.	Number of Procedures at Lex. Level I Relative to the Number at Level I-1				
	2*	3	4	5	>6
216	141	1.270	1.330	0.256	0.623
217	143	1.287	1.337	0.276	0.632
218	145	1.317	1.419	0.306	0.337
219	155	1.284	1.382	0.331	0.319
220	155	1.310	1.350	0.332	0.319
221	157	1.357	1.263	0.335	0.478
222	167	1.275	1.249	0.335	0.494
223	165	1.255	1.333	0.330	0.484
224	164	1.287	1.327	0.446	0.248
225	174	1.437	1.136	0.496	0.248
226	171	1.415	1.202	0.519	0.258
227	175	1.474	1.271	0.470	0.279
228	176	1.477	1.281	0.468	0.269
Overall Change (%)	+24.8	+16.3	-3.6	+82.8	-56.8

* All procedures at level 2 are declared within a single global outer block.

TABLE Va

Overall Evolution of the Procedure Calling Structure

Calling Level	Number of Procedures Called From This Level	
	Release 216	Release 228
1	12	13
2	55	79
3	211	289
4	699	1122
5	1596	3448
6	4025	8928
7	8283	22567
8	16956	52898
9	32868	113614
10	61187	216923
11	105623	368206
12	170146	574178
13	256743	838794
14	359281	1148985
15	471836	1458242
16	584529	1732477
17	683156	1955032
18	761001	2116502
19	808839	2183205
20	821540	2146774
21	800674	2038043
22	748287	1885601
23	671218	1706420
24	576174	1503538
25	472905	1291251
26	370764	1085380
27	276984	892888
28	197591	714108
29	134740	550749
30	88301	406415
31	55637	287039
32	34453	196318
33	20754	130141
34	12417	80514
35	7181	44250
36	3966	20137
37	1980	7499
38	888	2210
39	332	486
40	96	80
41	16	8

TABLE Vb

Distribution of Procedure Calls by Level and Release

Release Version No.	Percentage of the Cumulative Distribution of Procedure Calls to Level				
	10	15	20	25	30
216	1.30	15.47	53.49	87.46	98.56
217	1.27	15.45	53.90	87.61	98.54
218	1.21	15.17	53.70	87.63	98.63
219	0.94	13.83	53.33	88.79	99.16
220	1.45	18.47	59.62	90.40	99.00
221	1.19	15.70	55.43	88.99	99.05
222	1.47	18.01	58.43	90.08	99.12
223	1.83	19.96	60.78	90.85	99.18
224	1.32	17.60	59.50	90.96	98.87
225	1.36	17.30	57.93	89.54	98.66
226	1.32	16.22	55.05	87.53	98.24
227	1.52	17.24	53.53	83.93	97.18
228	1.51	17.30	53.77	84.09	97.23

TABLE VI

Percentage of Terminal Nodes by Level and Release

Release Version No.	Percentage of Nodes which are Terminal at Level							Deepest Level
	5	10	15	20	25	30	35	
216	33.8	49.3	59.5	62.6	63.4	65.6	66.5	41
217	33.1	49.2	59.7	63.6	64.9	66.8	68.3	41
218	32.6	48.1	58.6	63.0	64.9	66.8	68.7	41
219	32.1	46.4	57.5	62.9	65.0	67.7	69.4	40
220	32.1	47.6	59.4	63.6	65.1	66.9	70.7	39
221	31.9	47.3	58.8	63.3	65.0	67.3	70.7	39
222	31.4	48.6	58.2	61.9	64.4	66.8	70.9	39
223	31.1	47.8	58.0	63.0	65.6	67.7	71.0	39
224	31.5	45.9	54.2	58.8	62.7	64.5	69.0	41
225	33.7	46.5	54.4	58.7	62.2	64.5	69.5	40
226	32.2	46.9	54.3	58.5	61.8	64.2	68.4	41
227	31.4	47.4	54.9	58.6	60.7	63.3	69.4	41
228	31.6	47.4	55.2	59.0	61.1	63.7	69.6	41

TABLE VII

Distinct Procedure Calls by Level and Release

Calling Level	Distinct Calls at Level for Release Version No.						
	216	217	218	219	220	221	222
1	12	12	12	12	11	12	12
2	19	19	19	24	23	26	23
3	53	54	59	67	74	68	69
4	107	111	121	134	151	147	155
5	187	186	202	227	254	301	309
6	216	230	266	274	266	232	249
7	394	406	418	453	287	302	340
8	305	315	308	328	358	397	411
9	199	220	251	274	294	286	280
10	79	87	122	111	190	166	161
11	23	25	27	30	56	56	61
12	24	24	26	20	17	18	20
13	10	10	10	13	11	11	11
14	11	11	11	6	9	9	9
15	9	10	10		6	6	5
16	4	4	4				1
17							
18							

TABLE VII (Continued)

Calling Level	Distinct Calls at Level for Release Version No.					
	223	224	225	226	227	228
1	12	12	13	13	13	13
2	23	22	33	36	37	37
3	70	70	82	85	87	87
4	157	163	157	160	165	167
5	318	335	298	315	332	336
6	238	254	220	225	239	244
7	336	337	330	334	344	345
8	407	420	481	495	507	511
9	279	273	304	319	330	334
10	162	163	196	198	202	205
11	57	74	83	90	90	93
12	19	22	59	60	65	65
13	11	14	20	20	31	31
14	9	22	22	22	29	29
15	6	6	11	11	28	28
16		1	1	1	12	12
17			2	2	6	6
18					5	5

TABLE VIII

Evolution Of Some Complexity Parameters

Release Version No.	Total Distinct Calls	Complexity Coefficients			
		C_1	C_2	C_3	C_4
216	1652	7.06	54.9	463	4225
217	1734	7.08	55.2	466	4251
218	1866	7.12	55.8	472	4314
219	1973	6.97	53.1	432	3716
220	2007	7.14	56.4	478	4319
221	2037	7.10	55.6	469	4210
222	2116	7.08	55.2	464	4156
223	2104	7.06	55.0	461	4120
224	2190	7.13	56.5	486	4501
225	2312	7.36	60.3	537	5138
226	2386	7.35	60.2	536	5116
227	2522	7.50	63.7	598	6165
228	2548	7.51	63.7	598	6153
Overall Change (%)	54.2	6.28	15.82	29.46	48.10

TABLE IXa

Evolution of Sub-System WS PROCESS

Release Version No.	Total Distinct Calls	Complexity C_2		Coefficients C_3	
216	605	50.99		431.4	
217	613	51.79		442.3	
218	637	52.55		449.0	
219	689	51.99		441.3	
220	705	50.33		418.3	
221	711	50.03		415.2	
222	788	49.55		402.4	
223	751	48.99		397.6	
224	756	50.17		419.2	
225	762	51.12		433.5	
226	764	49.07		404.0	
(percent change)		-----	12.6%	-----	23.8%
227	858	55.27		500.5	
228	861	55.55		504.3	

TABLE IXb

Evolution of Sub-System FAMPRINT

Release Version No.	Total Distinct Calls	Complexity Coefficients	
		C_2	C_3
226	130	52.93	446.1
227	180	87.56	1023.2

TABLE X

Evolution of Sub-System OPSCOM

Release Version No.	Total Distinct Calls	Complexity Coefficients C_2	C_3
216	667	39.83	385.4
217	701	39.19	375.9
218	1710	52.31	461.6
219	1801	49.98	417.0
220	1842	62.60	562.5
221	1818	63.05	570.8
222	1913	64.80	594.4
223	1889	64.10	585.3
224	1968	63.51	575.7
225	928	47.79	423.6
226	1015	46.05	390.1
227	1042	45.08	378.0
228	1048	45.14	378.2

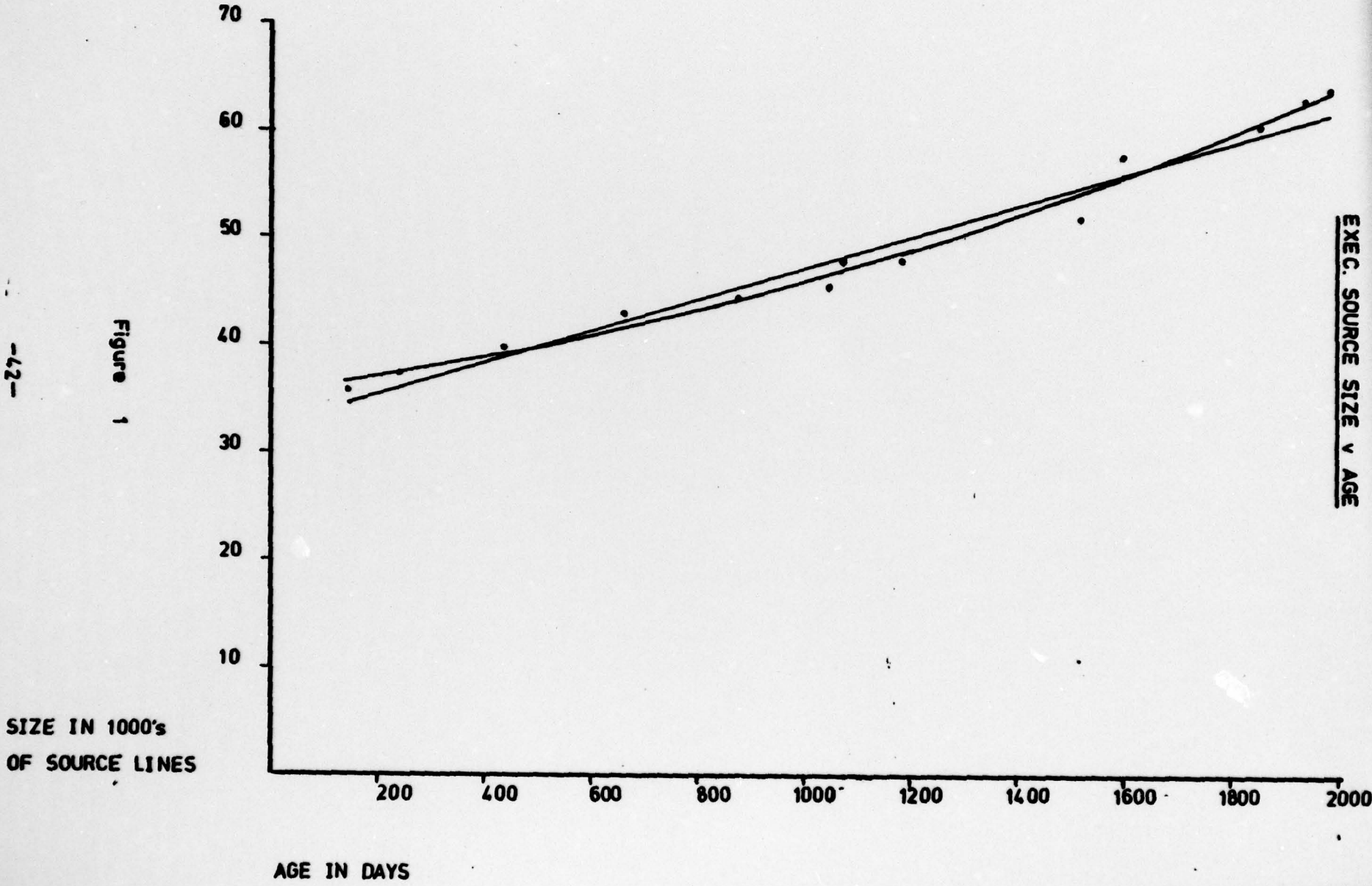

EXEC. SOURCE SIZE v AGE
SIZE IN 1000's
OF SOURCE LINES
70
60
50
40
30
20
10
200
400
600
800
1000
1200
1400
1600
1800
2000
AGE IN DAYS

Figure 1

EXEC. SOURCE SIZE v RELEASE NUMBER

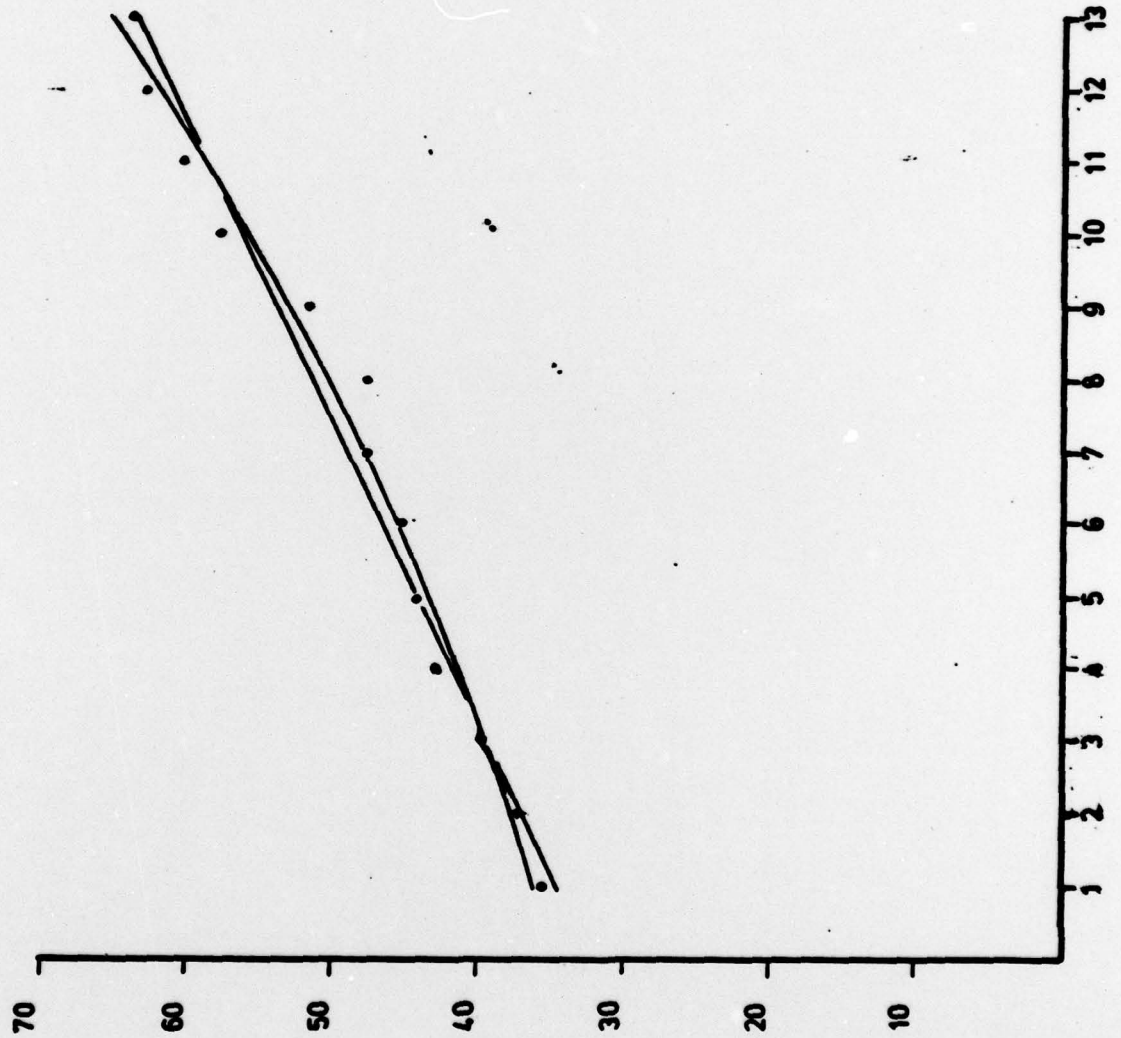


Figure 2

PROCEDURES v SOURCE LINES

Figure 3